X-QA
76
.6
A29

Keep
on
Hold
Shelf

1/22/76

RJF -

Here are my lecture notes for the MACSYMA IAP course. They may be more rightly termed "chicken scratches", but you can do with them what you may.

JPG

1976 IAP - MACSYMA course — Jan 5, 1976

Jeff Golden — Lecture 1, etc.

① Handouts: Primer, ITS memo, 1st prob. set.

Ask them to hand in a piece of paper with name, local address, field, level, programming experience, major reason for attending course.

Go over some of the mechanics of course.

② Basics of logging in to PDP-10, getting a MACSYMA, using it, killing it, logging out, sending mail, reading mail, getting help.

Log in as JRSIAP. Basic quit commands.

③ straightforward MACSYMA commands: SIN, COS, EXP, LOG, DIFF, INTEGRATE, LIMIT, SUM, MATRIX, DETERMINANT, SOLVE, EXPAND, FACTOR

④ Basic MACSYMA syntax + * - / ↑ ** ( ) ; $ =

C-lines, D-lines

Basic manipulations: SUBST, PART, EV

IF THEN ELSE, function defs., def. of FACTORIAL

e, i, π, :

⑤ Other introductory points: meaning of algebraic manipulation, facts about MACSYMA, LISP, size of MACSYMA.

Sequence: input of expression, parsing, evaluation and simplification, display.

1976 IAP - MACSYMA course — Jan. 7, 1976

Jeff Golden — Lecture 2, etc.

① Handout pgs. 2,
Primer, ITS memo for anyone who didn't get one the first class.

② Questions?

③ ITS issues: sending macsym mail, reading own mail, :KILL, deleting chars., quit commands, ??, editor

④ MACSYMA issues: ; assignment, some important commands:
PART, SUBST, EV (numer, exponentialize), PART (expr, 0),
function def. via MYFACTORIAL; IF THEN ELSE.

⑤ Sequence of manipulation:
input of expression, parsing, evaluation and simplification, display.

⑥ simplification = transforming an expression into a simpler i.e. more concise, more understandable, more useful form.
What is simpler depends on the circumstances, the use to which the expression is put.
general vs. cre (rational) representation

⑦ Assortment of one-step simplification commands:
EXPAND
FACTOR
RATSIMP $\frac{1}{x^2+3x+2} + \frac{x+1}{x+2} \rightarrow \frac{x^2+2x+2}{x^2+3x+2}$
RADCAN $\log(e^{2x} + 2e^x + 1) / \log(e^x + 1)$
MAP('RATSIMP, expr)
LAMBDA
SUBST vs. RATSUBST
COEFF vs. RATCOEF

⑧ For next class: Programming examples of MYTAYLOR, MYFIB, LEGENDRE, MYLIMIT
EV, ODE2, SAVE, lists, matrices, LINSOLVE.

1976 IAP - MACSYMA course — Jan 9, 1976

Jeff Golden — Lecture 3, etc.

① Handout prob. 3. Repeat mention of error in Prob. 2. Repeat schedule: A.I. Playroom 8th flr. orange rug.
Mention prob. 2 real prob., done in 1971, KdV eqn. MACSYMA begun in 1969. Say something about Prob. 3, 5.

② Reading own mail: U = yes. (no) = no. [can skip]

③ (D4) SIN(X); EXPONENTIALIZE; TRUE $; D4;

④ Review idea of simplification, general vs. ore representation.
SUBST vs. RATSUBST, COEFF vs. RATCOEF.
example: (a) RATSUBST(A, X*Y^2, X^4*Y^8 + X^4*Y^2); (out of manual)
$AX^3Y + A^4$ vs. what SUBST gives.

(b) COS(X)^4 + COS(X)^3 + COS(X)^2 + COS(X) + 1;
RATSUBST(1 - SIN(X)^2, COS(X)^2, %);
$\sin^4 x + \cos x\,(2 - \sin^2 x) - 3\sin^2 x + 3$ vs. SUBST

⑤ Assortment of one-step transformation (simplification) commands:
EXPAND, RATEXPAND, FACTOR, MULTTHRU, others TRIGREDUCE TRIGEXPAND

RATSIMP $\dfrac{1}{x^2+3x+2} + \dfrac{x+1}{x+2} \rightarrow \dfrac{x^2+2x+2}{x^2+3x+2}$

RADCAN $\log(e^{2x} + 2e^x + 1) / \log(e^x+1) \rightarrow 2$

MAP('RATSIMP, expr) expr = $\dfrac{x+1}{x^2+2x+1} + \dfrac{x+2}{x^2+5x+6}$

LAMBDA → don't mention [can skip]

⑥/⑦ lists, matrices, matrix multiplication, matrix inverse, DETOUT.

⑦ Programming in MACSYMA - function defs
(1) MYFACTORIAL; IF THEN ELSE. what if called on -1?
DEFINE

(2) MYTAYLOR: $f(x) \approx f(a) + (x-a)f'(a) + \frac{(x-a)^2}{2!} f''(a) + \cdots$

$+ \frac{(x-a)^n}{n!} f^{(n)}(a)$ {+ errorterm}

What can't the def. do?

$\frac{\sin(x)}{x}$, Laurent

(3) MYFIB: FIB[1] := 1 $ FIB[2] := 1 $ {MYFIB thwart}

FIB[N] := FIB[N-1] + FIB[N-2] $

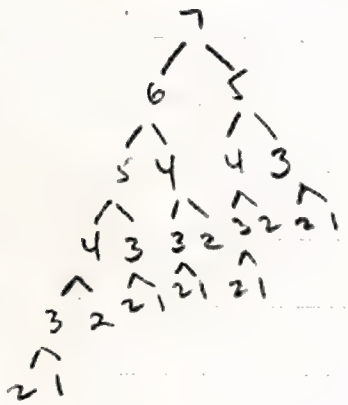
vs. MYFIB2(N) := IF N=1 OR N=2 THEN 1

ELSE MYFIB2(N-1) + MYFIB2(N-2) $

(4) LEGENDRE: Don't do.

(5) MYLIMIT.

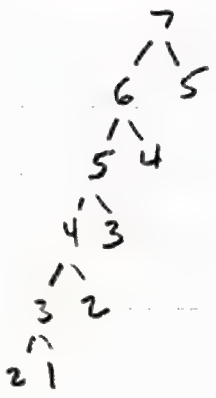
MYFIB2



6	1
5	2
4	3
3	5
2	8

vs.

MYFIB



1976 IAP - MACSYMA course — Lecture notes
Jeff Golden — for Lecture 3

Def. of MYTAYLOR

```
                    f(x)   x    a    n
MYTAYLOR(EXPR,VAR,POINT,HIPOWER) :=
    BLOCK([RESULT],
          RESULT: SUBST(POINT,VAR,EXPR),
          FOR I:1 THRU HIPOWER
          DO (EXPR: DIFF(EXPR,VAR)/I,
              RESULT: RESULT + (VAR-POINT)↑I *
                                SUBST(POINT,VAR,EXPR)),
          RETURN(RESULT)) $
```

Def. of MYFIB:

```
MYFIB[1]:1 $    MYFIB[2]:1 $
MYFIB[N]:=MYFIB[N-1]+MYFIB[N-2] $
```

Def. of LEGENDRE:

```
P[N]:=RATSIMP(1/(2^N*N!)*DIFF((X^2-1)^N,X,N));
```

/or/

```
Q[0]:RAT(1);        Q[1]:RAT(X);
Q[N]:=(2*N-1)/N*X*Q[N-1]
                - (N-1)/N*Q[N-2];
```

1976 IAP- MACSYMA course
Jeff Golden

lecture notes
for lecture 3

MYLIMIT lim expr, var → val, no infinities, two-sided

```
MYLIMIT(EXPR, VAR, VAL) :=
  BLOCK([OP],
   IF ATOM(EXPR)                                  ↙ SUBST(VAL, VAR, EXPR)
      THEN IF EXPR=VAR THEN RETURN(VAL)
                       ELSE RETURN(EXPR),
   OP: PART(EXPR, 0),
   IF OP = "+"
      THEN RETURN(MYLIMIT(FIRST(EXPR), VAR, VAL)
                  + MYLIMIT(REST(EXPR), VAR, VAL)),
      /or/
      THEN RETURN(MAP(LAMBDA([TERM], MYLIMIT(TERM, VAR,
                              VAL)), EXPR)),
   IF ...   {"*", "-", "/", "**", composition, infinities, e^x}
                  indeterminant forms

   ...
   RETURN('MYLIMIT(EXPR, VAR, VAL))) $
```

1976 IAP - MACSYMA course — Jan. 12, 1976

Jeff Golden — lecture 4

① Handout prob. set 3. Say something about prob. 3, 5 wrt their mathematical background. (Post defs. inside room 8003?)
Write on blackboard a solution to prob. 1.

② Mention subscript notation to access and assign elements of lists and matrices as an alternative to PART.

③ Programming in MACSYMA - function defs
<see notes for lecture 3, etc.>
Begin with review of MYFAC.

→ 2b SOLVE for systems of linear eqns.

(C1) SOLVE([X+Y=5, X-Y=1], [X, Y]);

(E1) X=3 (E2) Y=2 [[E1, E2]] GLOBALSOLVE

3b If time allows:

(manual p. 126)

```
NONØFREEOF(X,E):= IS(E#0 AND FREEOF(X,E)) $
MATCHDECLARE(A, NONØFREEOF(X), B, FREEOF(X)) $
DEFMATCH(LINEAR, A*X+B, X) $
e.g. LINEAR(3*Z+2, Z);
        [A=3, B=2, X=Z].
```

Also can do $(-1)^{ak+bn}$ pattern as done in my listing.

<over>

```
MATCHDECLARE ([A,B], INTEGERP, C, TRUE) $
DEFMATCH(PATT, A*K+B*N+C) $
PRED(X) := BLOCK([A,B,C],
                 PATT(X),
                 PANS: ABS(REMAINDER(A,2))*K
                       +ABS(REMAINDER(B,2))*N
                       +C,
                 RETURN(TRUE)) $
MATCHDECLARE(P,PRED) $
TELLSIMPAFTER((-1)↑P, (-1)↑PANS) $
```

e.g. $(-1)\uparrow(3*N+5*K)$; $\rightarrow$ $(-1)^{N+K}$

$(-1)\uparrow(2*N-6*K)$; $\rightarrow$ 1

Other issues worth talking about:

- Batch and Save files
- DEMO and SHARE directories.
- Access to ML/MC after course is over.

November 1975

Notes on Symbolic Manipulation
Hensel's Lemma

Hensel's lemma is an interpolation algorithm for "lifting" a factorization modulo a prime p to a factorization modulo a power of that prime. As we shall see, this approach can also be extended to obtaining a factorization in several variables over the integers from one in one variable modulo a small prime p.

Suppose we have the factorization

$$U(X) \equiv V_1(X)W_1(X) \text{ (modulo } p)$$

If we also have that

$$\gcd(V_1(X), W_1(X)) \equiv 1 \pmod{p}$$

then we can find $V_2(X), W_2(X), \ldots, V_k(X), W_k(X)$ such that

$$U(X) \equiv V_k(X)\, W_k(X) \pmod{p^k}, \qquad k > 1$$

A variant of Hensel's approach will allow one to obtain in k steps

$$U(X) \equiv V_k(X)\, W_k(X) \pmod{p^{2^k}}.$$

The latter approach is a quadratically converging algorithm (see notes on factorization, p. 398, problem 22).

The Hensel algorithm treats integers in their so-called p-adic representation. Thus if p=3, $65 \equiv 2\cdot 1 + 0\cdot 3 + 1\cdot 9 + 2\cdot 27$.

More generally, any positive integer n has a unique representation

$$n = a_0 + a_1 p + a_2 p^2 + \ldots + a_k p^k$$

where $0 \le a_i < p$

If we want to represent all integers we relax the condition on the a_i to $-p/2 \le a_i \le +p/2$.

We shall now show how to lift a mod p factorization to one mod p^2. Suppose

$$U(X) \equiv V_1(X)\ W_1(X) \pmod{p}$$

Let $C_1(X)$ be the polynomial determined as follows.

$$U(X) - V_1(X)\ W_1(X) \equiv pC_1(X) \pmod{p^2}$$

On the left-hand-side arithmetic is to be done mod p^2. The right-hand-side is guaranteed to be a multiple of p by the previous equation. The key idea now is to get "corrections" to V_1 and W_1 so that they encompass the polynomial $pC_1(X)$. This is possible because V_1 and W_1 are relatively prime.

If $V_1(X), W_1(X)$ are relatively prime (mod p) there exist unique polynomials mod p, A(X), B(X) such that

$$AV_1 + BW_1 \equiv 1 \pmod{p}$$

where degree A < degree W. Uniqueness is easy to show given the degree bound and is left as an exercise! For the examples we shall consider, A,B can be determined by inspection. In general they can be found from the Extended Euclid's ~~GCD~~ algorithm.

If we multiply the equation above by $C_1(X)$, we get

$$AC_1 V_1 + BC_1 W_1 \equiv C_1 \pmod{p}$$

The polynomial AC_1, BC_1 would work as correction for a while, but in general would give us trouble because their degrees would be too high in X. Once again there exist unique polynomials A^* and B^* such that

$$A^*V_1 + B^*W_1 \equiv C_1 \pmod{p}$$

when degree A^* < degree W_1, whenever degree C_1 < degree U(X). A^* is easily found to be the remainder after dividing AC_1 by W_1. B^* is then determined to fit the equation.

Now we correct V_1 and W_1 as follows.

$$V_2 = V_1 + p\,B^*$$
$$W_2 = W_1 + p\,A^*$$

Thus

$$\begin{aligned} V_2W_2 &= V_1W + p(A^*V_1 + B^*W_1) + p^2V_1W_1A^*B^* \\ &= V_1W_1 + p(A^*V_1 + B^*W_1) \pmod{p^2} \\ &= U(X) \pmod{p^2} \end{aligned}$$

The following example uses two steps of the "linear" Hensel algorithm.

$$U(X) = X^2 + 27X + 176, \text{ with } p=3$$
$$U(X) \equiv X^2 + 2 \ (3)$$

Using Berlekamp's algorithm, we get

$$U(X) \equiv (X+1)(X+2) \ (3)$$

The first correction term is

$$(X^2 + 27X + 176) - (X^2 + 3X + 2) = 24X + 176 \equiv 6X + 3 \ (9)$$
$$6X + 3 = 3(2X+1) \ (9)$$

Hence $$C_1(X) = 2X + 1$$

Note $$-1(X+1) + 1(X+2) \equiv 1 \ (3)$$

Hence $$2(X+1) + 1(X+2) \equiv 1 \ (3)$$

and $$(4X+2)(X+1) + (2X+1)(X+2) \equiv 2X+1 \ (3)$$

Dividing $4X+2 \equiv X+2$ by $X+2$ yields a remainder of 0.

Hence $$0 * (X+1) + 2(X+2) \equiv 2X+1 \ (3)$$

Therefore $$V_2(X) = X+1 + 2 * 3 = X+7 \ (9)$$
$$W_2(x) = X+2 + 0 * 3 = X+2 \ (9)$$

Now lift from p^2 to p^3

$$\begin{aligned} (X^2+27X+176) - (X+7)(X+2) &\pmod{27} \\ = 18X + 162 \equiv 18X &\pmod{27} \\ = 9\,(2X) &\pmod{27} \end{aligned}$$

$$2(X+1) + 1(X+2) \equiv 1 \ (3)$$
$$4X(X+1) + 2X(X+2) \equiv 2X \ (3)$$

Now $4X \equiv X \ (3)$ has remainder 1 after division by $X+2$.

$$1(X+1) + 1(X+2) \equiv 2X \ (3)$$

Therefore $V_3 = (X+7) + 9 * 1 = X+16$ (27)
$W_3 = X+2 + 9 * 1 = X+11$ (27)

$$(X^2+27X+176) - (X+16)(X+11) \equiv 0 \quad (81)$$

and we are done.

The most amazing aspect of Hensel's lemma is that we can use it to obtain a factorization in several variables. This means that under favorable circumstances factorization of a polynomial in, say, 20 variables, requires one to factor a polynomial in one variable and lift the factorization at a cost of a few multivariate multiplications.

Kronecker, of course, had other ideas. It might be instructive to see his approach which uses classical interpolation. Kronecker's idea is to evaluate all but one variable at integer points and construct the unique polynomial coefficients in the other variables from the factors that result. For example, consider

$$(Y+1)X^2 + (Y^2+1)X + 2Y-2$$

at Y=0 $$X^2+X-2 = (X+2)(X-1)$$

at Y=1 $$2X^2+2X = (2X+2)\ X$$

(choose this order to make this algorithm work better)

at Y=2 $$3X^2+5X+2 = (3X+2)(X+1)$$

Now interpolate coefficient by coefficient to obtain the quadratic (at worst) polynomials in Y.

1,2,3	→	Y+1
2,2,2	→	2
1,1,1	→	1
-1,0,+1	→	Y-1

Hence the factorization is $((Y+1)\ X + 2)(X + (Y-1))$.

Kronecker's approach loses for two reasons:

a) He requires a univariate factorization at each step.

b) If the original polynomial had k variables of degree d each, the algorithm would require $(d+1)^k$ factorizations. For k=20, d=1 only, one needs about a million factorizations. The polynomial could be as simple as $X^2+a+b+c+...+t-1$.

Hensel's approach in this case works very much like the modulo p case. However, the other variables act as moduli as well.

Let Y=0 in the example above to get

$$X^2+X-2 = (X+2)(X-1) \pmod 5$$

This can also be considered to be a factorization mod Y as well. Thus we are looking for a factorization mod Y^2. That is, we are looking for a factorization such as

$$U(X,Y) = V_2(X,Y)W_2(x,Y) \pmod{5,Y^2}$$

where V_2, W_2 are mod Y^2 and hence linear in Y.

Therefore consider

$$U(X,Y)-V_1(X)W_1(X) = \text{linear terms in } Y = YX^2+2Y$$
$$= Y(X^2+2) \pmod{5,Y^2}$$

To correct X^2+2 we go through the same process as before

$$2(X+2) + 3(X-1) \equiv 1 \quad (5)$$

The remainder of X^2+2 when divided by X-1 (mod 5) is 1.

Hence $$1(X+2) + X(X-1) = X^2+2 \quad (5)$$

Thus
$$X+2 + Y * X = X(Y+1) + 2$$
$$X-1 + Y * 1 = X + Y-1$$

and we are done.

The situation with more than two variables is similar. Every term $g(X)Y_1^{n1}\ Y_2^{n2} \ldots Y_m^{nm}$ is treated separately, however.

An example will be instructive.

Consider
$$U(X,Y,Z) = (YZ + 1)X^2 + (Y^3Z+Y^2+YZ^2+2YZ+2Z+3)X$$
$$+(Y^2Z+Z^2+Y+3Z+2)$$

$$= (X+Y^2+Z+2)((YZ+1)X+Z+1)$$

let Y=0, Z=0
$U(X,0,0) = V_1(X)W_1(X) = X^2+3X+2 = (X+1)(X+2)$
$= (X+1)(X+2) \pmod 5$

To get the linear terms in Y and Z, first obtain the difference modulo Y^2, YZ, Z^2, the quadratic terms. The difference is precisely $2ZX+3Z = Z(2X+3)$. We need thus to correct for 2X+3

$$-1(X+1) + 1(X+2) \equiv 1 \ (5)$$
$$1(X+1) + 1(X+2) \equiv 2X+3 \ (5)$$

Thus

$$X+1 + Z*1 = X+Z+1$$
$$X+2 + Z*1 = X+Z+2$$

are the the factors mod the quadratic terms. Multiplying these out and subtracting again we get the difference $YZ(X^2+2X) + Y^2(X+1)$ which we have grouped about the different quadratic terms.

X^2+2X can be obtained as

$$0(X+1) + X(X+2) \equiv X^2+2X \ (5)$$

X+1 can be obtained as

$$1(X+1) + 0(X+2) \equiv X+1 \ (5)$$

Therefore the factors are

$$X+Z+1 + YZ*X$$
$$X+Z+2+Y^2$$

and we are done after checking the product.

There are cases when the straight-forward use of Hensel's lemma does not quite work. For example, if Y=0 causes the leading coefficient in X to go to 0, we will not obtain a correct factorization for X. In such a situation we need to substitute, for example, Y=1 and consider U(X,Y-1,...). This is not so bad unless we need several such substitutions. For example, Y=1, Z=1 will cause the single term $Y^{10}Z^{10}$ to explode to

$$(Y-1)^{10}(Z-1)^{10}$$

which has about 100 terms. In such a case we are back to exponential behavior. It is currently suspected that there is no general way of getting around this "bad zero" problem (which fortunately does not occur that often). Another problem is the so-called "leading coefficient problem". For a way of getting around this see the notes p. 397, problem 18.

for RJF

MACSYMA IAP course 1976

1st Problem

Consoles are in 800D. Sign up for at most 2 hours during the 9:00 to 5:00 time period and no more than 2 hours at other times.

Integrate:

$$e^x \cos(x) \sin^2(x)$$

differentiate the result

simplify to obtain the same form as the input.

Due: Monday, January 12.

Please note: If you have any problems with the consoles or they seem to be out of order, please report this to the secretary in room 828.

You will receive free of charge copies of the "MACSYMA Primer" and "An Introduction to ITS for the MACSYMA User". If you wish a copy of the "MACSYMA Reference Manual", version 8, you may purchase one for $4.00 from Mrs. D. Kontrimus in room 417.

MACSYMA IAP course 1976

2nd Problem

due Wednesday, January 14

Show that

$$F = G(G-2)^2 e^{G\cdot Y} + 4 \frac{d^2}{dY^2} (e^{(G-1)Y} \operatorname{sech} Y)$$

is a solution to the differential equation

$$\frac{d^3F}{dY^3} - 4 (1-3 \operatorname{sech}^2 Y) \frac{dF}{dY} - (24 \operatorname{sech}^2 Y \tanh Y + A) F = 0$$

subject to the relation

$$G^3 - 4G - A = 0$$

MACSYMA IAP course 1976
A solution to the 2nd problem
(Run on MIT-MC, the faster PDP-10)

```
(C2) BATCH(KDV,5,DSK,JPG);

(C3) TIME:ALL$
TIME= 1 MSEC.

(C4) EXPONENTIALIZE:TRUE$
TIME= 1 MSEC.

(C5) F:G*(G-2)^2*EXP(G*Y)+4*DIFF(EXP((G-1)*Y)*SECH(Y),Y,2);
TOTALTIME= 179 MSEC.  GCTIME= 4 MSEC.
```

$$
\begin{aligned}
(D5)\quad & (G-2)^2\, G\, \%E^{G\,Y} + 4\left(\frac{2\,(G-1)\,\%E^{2\,(G-1)\,Y}}{\%E^{Y}+\%E^{-Y}} - \frac{2\,\%E^{(G-1)\,Y}}{\%E^{Y}+\%E^{-Y}}\right. \\
& \left. - \frac{4\,(G-1)\,(\%E^{Y}-\%E^{-Y})\,\%E^{(G-1)\,Y}}{(\%E^{Y}+\%E^{-Y})^2} + \frac{4\,(\%E^{Y}-\%E^{-Y})^2\,\%E^{(G-1)\,Y}}{(\%E^{Y}+\%E^{-Y})^3}\right)
\end{aligned}
$$

```
(C6) DFDY1:DIFF(F,Y);
TIME= 241 MSEC.
                                                        (G - 1) Y
             2  2   G Y          6 (G - 1) %E
(D6) (G - 2)  G  %E    + 4 (- --------------------
                                      Y     - Y
                                   %E  + %E

             3   (G - 1) Y             2    Y     - Y    (G - 1) Y
   2 (G - 1)  %E            6 (G - 1)  (%E  - %E   ) %E
 + ---------------------- - -----------------------------------
           Y     - Y                        Y     - Y 2
         %E  + %E                         (%E  + %E   )

          Y     - Y    (G - 1) Y
   10 (%E  - %E   ) %E
 + ---------------------------
             Y     - Y 2
           (%E  + %E   )

                  Y     - Y 2   (G - 1) Y
   12 (G - 1) (%E  - %E   )  %E
 + ------------------------------------
                  Y     - Y 3
                (%E  + %E   )

          Y     - Y 3   (G - 1) Y
   12 (%E  - %E   )  %E
 - ----------------------------)
             Y     - Y 4
           (%E  + %E   )

(C7) DFDY3:DIFF(DFDY1,Y,2)$
TOTALTIME= 1388 MSEC.  GCTIME= 367 MSEC.

(C8) A:G^3-4*G$
TIME= 6 MSEC.

(C9) DE:DFDY3-4*(1-3*SECH(Y)^2)*DFDY1-(24*SECH(Y)^2*TANH(Y)+A)*F$
TIME= 63 MSEC.

(C10) KILL(LABELS,F,A,DFDY1,DFDY3)$
TIME= 4 MSEC.

(C1) RADCAN(DE);
TOTALTIME= 7474 MSEC.  GCTIME= 2701 MSEC.
(D1)                                0

TOTALTIME= 10623 MSEC.  GCTIME= 3531 MSEC.
(D2)                            BATCH DONE

(C3) CLOSEFILE(KDV5,OUT);
```

MACSYMA IAP course 1976

Problems

due Friday, January 16

Problem 3

Using MACSYMA lists $[A_1, A_2, A_3]$ to represent 3-dimensional vectors $\underline{A} = A_1 \underline{i} + A_2 \underline{j} + A_3 \underline{k}$.

(a) Define MACSYMA functions DOT($\underline{A}$, $\underline{B}$) and CROSS($\underline{A}$, $\underline{B}$) which return the dot (scalar) and cross (vector) product of two vectors $\underline{A}$ and $\underline{B}$, respectively.

(b) Use your definitions to prove that

$$\underline{A} \times (\underline{B} \times \underline{C}) = (\underline{A} \cdot \underline{C})\,\underline{B} - (\underline{A} \cdot \underline{B})\,\underline{C}$$

for arbitrary vectors $\underline{A}$, $\underline{B}$, $\underline{C}$.

(c) Define MACSYMA functions GRAD(F), DIV($\underline{V}$), and CURL($\underline{V}$) which return the gradient of a scalar function F(X,Y,Z), and the divergence and curl of a vector function $\underline{V}$(X,Y,Z) respectively.

(d) If $F(X,Y,Z) = 3X^2Y - Y^3Z^2$, show that grad (F) at the point (1,-2,-1) is in MACSYMA representation [-12,-9,-16].

(e) If $\underline{A}$ is the vector function $[X^2, -2Y^3Z^2, XY^2Z]$ in MACSYMA representation, show that DIV($\underline{A}$) at the point (1,-1,1) is -3.

(f) If $\underline{A}$ is $[XZ^3, -2X2YZ, 2YZ^4]$, show that CURL($\underline{A}$) at the point (1,-1,1) is [0,3,4].

Problem 4

Without using the MACSYMA function DIFF, define a function MYDIFF(expr, var) which differentiates an expression *expr* with respect to a variable *var*. MYDIFF should be able to handle arbitrary polynomials in the given variable, say X, as well as sums and products of such polynomials with e^X, log X, SIN(X), and COS(X), and any other cases you wish to handle (one interesting possibility is the composition of the above functions using the chain rule, e.g. to handle MYDIFF(COS X^2,X)). E.G. it should be able to handle

MYDIFF(aY^4 cos Y sin Y + $4Y^3$ e^Y log Y + bY + c, Y)

Problem 5

Using MACSYMA, show that the matrix

$$A = \begin{pmatrix} 2 & \sqrt{2} \\ \sqrt{2} & 1 \end{pmatrix}$$

has eigenvalues $\lambda_1 = 0$, $\lambda_2 = 3$ and possible orthonormal eigenvectors

$u_1 = [-1/\sqrt{3}, \sqrt{2/3}]$, $u_2 = [\sqrt{2/3}, 1/\sqrt{3}]$.

MACSYMA IAP COURSE
January 1976

Problem 2

(g) Define a MACSYMA simplification rule to encode the fact stated in part b, namely:

$$A \times (B \times C) = (A \cdot C)\, B - (A \cdot B)\, C$$

where A, B, C are arbitrary, unspecified vectors (ie just symbols, not lists).

R. Moenck
U. waterloo
CS75

§1 The Radical Approach to Algebraic Manipulation

In algebraic manipulation we would like to perform operations on classes of functions that might include:

C = {polynomials, rational functions, rational integers and their algebraic extensions, and a range of transcendental functions}

We are not just interested operations on these functions alone but also in operations on the class of functions produced by composing various subsets of this class C.

If we try and take on C* at one gulp we quickly get into confusing detail. It is better to take what we will call a "radical" approach, which is to consider only a small subclass of the functions and then gradually add other functions from C into it. In this way we can give a consistant treatment of the issues.

We can use as a lantern to light our way, the classical results of "modern" algebra (as seen in Van der Waerden, or Birkhoff and MacLane). The constructive approach to algebra as used in the nineteen twenties and thirties is more suited to the development of algorithms than the more recent existential approach.

So initially we will consider polynomials over a ring:

$$R\,[x_1][x_2];\ldots,[x_n]$$

We will start with polynomials since they are fairly well understood mathematically and because they are familiar objects which people have worked with since high school. In our polynomials $\{\bar{x}\}$ will be the indeterminants (variables) and R will be some number ring eg.

$\underline{Z}$ (the integers) or $\underline{Q}$ (the rationals)

or $\underline{Z}_m$ (the integers) mod m or some more

exotic number ring like $\underline{Z}(\pi)$ (the field of rational functions of π).

1.1) Representation Questions:

Since we are interested in actually specifying computer algorithms for these objects we must consider in more detail the structure and representation of them than would a purer mathematician. In general we can distinguish three levels of refinement:

1) logical level: where we consider the object itself

e.g. $a(\bar{x}) \in R[\bar{x}]$.

2) form level: where we consider how the elements of the object relate to one another.

e.g. consider $a(\bar{x})$ as a sum of terms.

3) Structural level: where we consider the representation of $a(\bar{x})$ in the computer.

e.g. as a linked list structure.

1.2) Operation Counts

Algebraic manipulation is generally a very costly process both in terms of computing time and the amount of storage used. Therefore we want to keep a close eye on the efficiency of our algorithms. One way we can do this is to analyse the algorithms with respect to the number of operations they use. If we can get a closed form expression for the number of operations then we have a basis for comparing the relative efficiencies of two competing algorithms.

So we make:

Defn. 1.1:

Any algorithm $\mathcal{A}$ has a certain well defined set I of valid inputs.

If an algorithm is initiated with a valid input x, the algorithm performs a certain finite number $T_A(x)$ of primitive actions (eg. computer instructions) and finally stops producing some output. $T_A(x)$ is the computing time function of the algorithm A.

To estimate the computing time we will use the dominance notation which is somewhat more precise than the O-notation as used by Knuth and others.

Defn. 1.2: If f and g are two real valued functions defined on some set S we say f is dominated by g or g dominates f

ie. $f \leq g$ if $\{c \mid c \in \underline{R}^+, |f(x)| \leq c|g(x)|, \forall x \in S\}$.

Defn. 1.3: If $f \leq g$ and $g \leq f$, we say that f and g are codominant

ie. $f \sim g$

and finally

Defn. 1.4: If $f \leq g$ but $g \not\leq f$ then we say g strictly dominates f ie. $f < g$.

§2 Operations on Integer Coefficients

Before we consider operations on polynomials we must look at their proper subring, their coefficients. In general the philosophy of algebraic manipulation is that of exact calculation and therefore the use of approximation as seen in floating point arithmetic is avoided. This means that the numerical analyst's headaches of rounding errors are swept aside but at the cost of having to deal with very lengthy coefficients. Since we allow only exact arithmetic it is very easy to do calculations which exceed the eight or ten decimal digits allowed for single precision integer operations by most computers. This means that we must consider multi-precision integer operations.

2.1 The Classical Method

The most common form of multiprecision integers is large base notation. If we choose some base β a multiprecision integer A can be represented as:

$$A = \sum_{i=0}^{n} a_i \beta^i \text{ where the digits, } 0 \le a_i < \beta.$$

In order to save storage β is usually chosen to be a number which fills a computer word. To analyse the algorithms used to operate on such integers we can define:

Defn. 2.1: The β-length $L_\beta(A)$ of an integer A is the number of β digits used to represent the integer. We can now use the same algorithms taught in elementary school (where $\beta = 10$) for performing operations on multiprecision integers.

This gives us:

Lemma 2.1 The time to add or subtract two integers x and y is:

$$T_A(x,y) \le \max (L(x), L(y))$$

$$\sim L(x) + L(y)$$

Lemma 2.2: The time to multiply two integers

$x,y \neq 0$ is:

$$T_M(x,y) \lesssim L(x)\cdot L(y) \quad .$$

2.1.1 Integer Division see, instead, Knuth, Vol 2 p 237 § 4.3.1 Algorithm D

The elementary school methods for simple integer operations are well specified except for division. In division the method for generating the successive digits of the quotient involves some human judgement, which must be eliminated in a computer algorithm. In 1960 Pope and Stein proposed the following algorithm.

Algorithm: Integer-division (U,V,Q,R);

Input: dividend U and divisor $V = \sum_{i=0}^{n} v_i \beta^i$

Output: quotient $Q = \sum_{i=0}^{k} q_i \beta^i$ and remainder R;

Steps:

1) Normalise divisor and dividend so that the leading digit of the divisor is at least $[\beta/2]$:

$$U = U\cdot [\beta/(v_n + 1)]; \quad V' = V\cdot [\beta/(v_n + 1)];$$

2) Iteration: Assume quotient digits $q_k,\ldots,q_{j+1}$ have been determined and $A = \sum_{i=0}^{n+j+1} a_i \beta^i$ is the current remainder:

Then q_j is approximated by $\bar{q}_j = [(a_{n+j+1}\beta + a_{n+j})/v'_n]$ unless $a_{n+j+1} = b_n$ in which case $\bar{q}_j = \beta - 1$.

Pope and Stein showed that in all case $0 \leq \bar{q}_j - q_j \leq 2$.

Thus at most two corrections are required to obtain q_j and the necessity of a correction is characterised by a negative remainder. □

This gives us:

Lemma 2.3: The time to divide two integers x by y obtaining a quotient q and remainder r is:

$$T_D(x,y,q) \lesssim L(y)\cdot L(q)$$
$$\sim L(y)\cdot(L(x) - L(y) + 1), \text{ for } |x|\geq|y|>0.$$

2.1.2) Greatest Common Divisors

The classical algorithm for computing the gcd of two integers is the familiar Euclidean algorithm (see section 4). This gives

Lemma 2.4: The time to compute $w = \gcd(x,y)$ $x \geq y > 0$ by the Euclidean algorithm is:

$$T_E(w,x,y) \leq L(x)\cdot L(y/w)$$
$$\sim L(b)\cdot(L(a) - L(c) + 1).$$

D.H. Lehmer has given a version of the Euclidean algorithm which is faster for larger integers. Lehmer's version has the same computing time function as the ordinary Euclidean algorithm to within codominance, but it is eight to ten times faster on a typical computer when the inputs are longer than one β-digit.

GCDs and the Euclidean Algorithm E have been extensively studied by number theorists. Some of these results apply to our interests.

Defn. 2.2: Let

$$S(m,n,k) = \{(w,x,y)\,|\,|x|\geq|y|>0,\ L(x) = m, L(y) = n \text{ and } L(w)=k \text{ where } w = \gcd(x,y)\}.$$

The set S(m,n,k) are finite, disjoint and together contain all valid inputs to E.

Defn. 2.3: The maximum computing time function for E

$$T_E^+(m,n,k) \triangleq \max \{T_E(w,x,y) \mid (w,x,y) \in S(m,n,k)\}$$

The result about T_E above can be restated as:

Lemma 2.5: $T_E^+(m,n,k) \leq n(m-k+1)$

In fact G. Collins has shown:

Theorem 2.1: $T_E^+(m,n,k) \sim n(m-k+1)$.

Analogously we can define:

Defn. 2.4: The minimum computing time function $T_E^-(m,n,k)$ for E.

$$T_E^-(m,n,k) \triangleq \min \{T_E(w,x,y) \mid (w,x,y) \in S(m,n,k)\}$$

Collins also shows:

Theorem 2.2: $T_E^-(m,n,k) \sim n(m-n+1) + k(n-k+1)$.

This implies that $T_E^- < T_E^+$.

We can also define

Defn. 2.5: The average computing time function for E

$$T_E^*(m,n,k) = \frac{\Sigma\{T_E(w,x,y) \mid (w,x,y) \in S(n,m,k)\}}{\text{Card } (S(m,n,k))} .$$

Collins has also shown:

Theorem 2.3: $T_E^* \sim T_E^+$.

This result is obtained using a result of Dixon's about the average number of divisions in the Euclidean Algorithm.

2.3) Representation of Multiprecision Integers

Now we turn to the structural level of representation of a multiprecision integers. There are several alternative methods.

a) Linked lists:

The SAC-1 system (among others) employs linked lists to store an integer. A list element contains (at least) two fields:

i) a successor field which contains the address of the next list element or a null pointer λ if there is no next link.

ii) an element field which contains the β-digit of the list element.

e.g.

For example if $\beta=10$ and to store the number A=352 we might have the structure.

Collins quotes the following times for SAC-1 on a UNIVAC 1108.

For $L_{10}(A) = L_{10}(B) = L_{10}(C) = 100$

and

$$L_{10}(D) = L_{10}(E) = 200$$

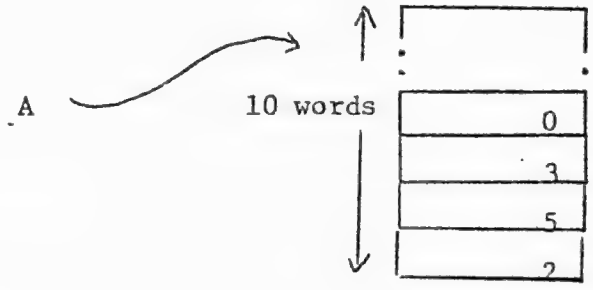
Table 2.1: Computing Times in seconds for Integer Arithmetic

C = A + B	0.001
D = A ∘ B	0.007
B = D / A	0.018
A = gcd(D,E)	0.150

b) Array Allocation

The Altran system (among others) allocates a fixed number k (usually 10) of array elements to store the β-digits of an integer.

eg. for β=10 and A=352



The advantage of this representation over the previous ones is that array indexing is usually faster than list element indexing as used above. The disadvantage is that the maximum length of integers is restricted to k β-digits whereas the linked system is limited only by the amount of computer memory. Another disadvantage is that considerable amounts of computer memory is wasted by storing 0 leading digits.

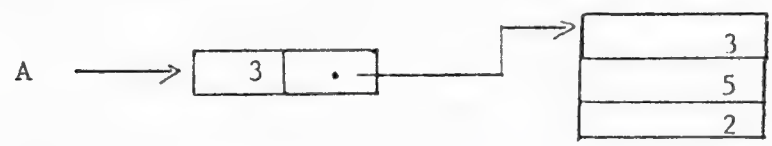
c) Descriptor Allocation:

An alternative which combines the best features of both systems is to use a descriptor of the allocation. The descriptor contains two fields

i) the length field containing the number of computer words used in the multiprecision integer

ii) the address field which contains the address of the first word in the array of β-digits.

eg. for β = 10 and A = 352



This system combines the best features of both systems. However it does require a more sophisticated garbage collection mechanism to reclaim

freed storage than the others.

2.4 Faster Operations

The literature of computational complexity contains a number of algorithms for multiprecision integer operations. We mention them here for the sake of completeness; in general, these algorithms are only more efficient for very large integers, larger than those normally encountered in algebraic manipulation.

There is a hierarchy of multiprecision integer multiplication algorithms, the currently fastest is one due to Schönhage and Strassen.

Theorem 2.4 Two integers x,y with $L(x) = L(y) = N$ can be multiplied in time:

$$T_{SS}(N) \leq N \log N \log \log N.$$

S. Cook has given a construction whereby multiprecision integer division can be performed in the same time as multiplication using a form of newton iteration. This gives:

Theorem 2.5 An integer x can be divided by an integer y where $L(x) = 2N$ and $L(y) = N$ in time:

$$T_C(N) \leq N \log N \log \log N.$$

A. Schönhage has given a method whereby integer gcds can be computed by taking advantage of fast integer multiplication algorithms. This leads to

Theorem 2.6 The gcd of two integers x and y where $L(x) = L(y) = N$ can be computed in

$$T_S(N) \leq N \log^2 N \log \log N.$$

The logarithms used in the above results are all base 2.

2.5 Other Structures for Integers

So far we have considered only a particular form for long integers namely that of base B arithmetic. While this is adequate for many purposes it has the disadvantage that some of the important operations (i.e. multiplication and forming gcds) require times (practically) which are quadratic in the length of the integers. There are other forms which permit faster algorithms for these operations at the expense of slower algorithms for other operations.

We will now look at 2 of these alternative methods.

2.5.1 Modular Arithmetic

Modular arithmetic exploits the ring homomorphism:

Definition 2.5.1: The Modular Homomorphism

$$\emptyset_m : \underline{Z} \to \underline{Z}_m, \qquad m \in \underline{Z}$$

$$\emptyset_m : a \mapsto a+(m) \equiv a \bmod m$$

e.g. $23 \equiv 3 \bmod 4$ and $23 \equiv 5 \bmod 9$.

We will consider this homomorphism in more detail in the context of other homomorphism algorithms (sect 5). For the moment it suffices to note that since this is a ring homomorphism it has the properties:

i) $\emptyset(a) + \emptyset(b) = \emptyset(a+b)$

ii) $\emptyset(a) \circ \emptyset(b) = \emptyset(a \circ b)$

and by virtue of the Chinese Remainder Theorem:

iii) $a \in \underline{Z}$ is uniquely represented by $\{a_i\}$

where $a_i = a \bmod m_i$

and $\pi m_i > 2|a|$ and $\{m_i\}$ are relatively prime (i.e. $\gcd(m_i, m_j) = 1$ when $i \neq j$).

Given a set of integer remainders and their corresponding moduli the original integer can be computed using the integer chinese remainder algorithm. The best practical algorithms for this are quadratic in nature (see Garner's Algorithm in section 5). However, there do exist faster algorithms for larger integers.

The ring homomorphism means that if we have two integers A, B with residues $\{a_i\}$, $\{b_i\}$ with respect to the same set of moduli $\{m_i\}$ where the product of the moduli is greater than their sum or difference, then we can form their modular sum C (or difference) by forming

$$c_i = a_i \pm b_i \mod m_i \quad \forall i.$$

This gives us a linear addition algorithm.

Usually the moduli are chosen to be close to the word size of the computer, i.e. $m_i \sim \beta$. This means that the number of moduli needed to represent an integer will be approximately equal to its length.

Lemma 2.5.1 The time to add two modular integers a, b.

$$T_{MA}(a,b) \sim \max(L(a),L(b)).$$

Correspondingly if we have sufficient residues of a and b to represent their product c then we can form the product by:

$$c_i = a_i \circ b_i \mod m_i \quad \forall i.$$

This gives us a linear multiplication algorithm

Lemma 2.5.2 The time to multiply two modular integers a, b is:

$$T_{MM}(a,b) \sim (L(a)+L(b)).$$

Similarly, if we wish to perform exact division (i.e. we know that the remainder is zero). Then

$$c_i = a_i \circ b_i^{-1} \mod m_i \quad \forall i,$$

which gives us a linear exact division algorithm. (note computation of b^{-1})

Lemma 2.5.3 The time to divide two modular integers a by b is:

$$T_{MD} \sim \max(L(a),L(b)).$$

These fast algorithms illustrate the advantages of this form. Now we mention some of the disadvantages:

i) Sign testing

It can be shown that to find the sign of a modular integer it is necessary to construct its β-base (or equivalent) counterpart.

ii) Division with Remainder

Related to this is the fact that there is no known algorithm for division with remainder or computing gcds of modular integers which do not involve reconstructing the β-base counterparts.

iii) A Priori Bounds

Another constraint is that one must know before performing an algorithm how many residues will be needed to represent the answer uniquely.

iv) Dynamic Residues

In part (iii) is due to the fact that there is no fast method for computing a new residue for a new modulus, given an existing modular integer, (i.e. no method other than computing the β-base or equivalent form).

v) Waste of Storage

In order to perform operations on a large and a small modular integer, sufficient residues must be used to represent the resulting modular integer. This might involve a very storage wasteful representation of the smaller integer. Martin describes a method whereby some of this problem can be overcome.

2.5.2 Factored Form

Another form which has seen less exposure is to represent a multiprecision integer by the powers of its prime factors. This representation depends on

Theorem 2.5.2 (The fundamental theorem of arithmetic)

Any integer a can be uniquely decomposed into a product of prime powers.

$$a = \prod_{i=0}^{n} P_i^{e_i}, \text{ where } P_i = \{2,3,5,7,\ldots\}.$$

By virtue of this we can uniquely represent an integer by the powers $\{e_i\}$ with their respective primes implicit.

The reason this form has not seen much use is that computing the factors of a multiprecision integer is a time consuming operation requiring up to $\sqrt{a}$ multiprecision divisions using a naive method. This gives

Lemma 2.5.4 A multiprecision integer a can be factored into its prime powers in:

$$T_F(a) \leq \beta^{L(a)/2} \cdot L(a)^2.$$

However, once we have an integer in factored form then multiplying dividing and finding gcds of such forms becomes trivial.

Definition 2.5.2 Let the power length P(a) be the length of the sequence of powers needed to represent an integer.

Then we have

Lemma 2.5.5 The time to multiply, divide exactly or form the gcd of two factored integers a, b is

$$T_{MF}(a,b) \sim \max(P(a),P(a)).$$

The topology induced on $\underline{Z}$ by P(a) is very different from that induced by L(a) and so they are almost incomparable. However, it seems that in general P(a) < L(a). A deep result of Ramanujan's is applicable.

Theorem 2.5.3 Any integer a has an average log log a prime factors.

In addition, in many computations the growth of integer coefficients is by some constant amount. This growth would be simplified and would lead to more understanding of the problem if a factored representation were used. Also this form can be easily applied to representing rational numbers if negative exponents are allowed.

The principle disadvantage of this form is that addition and division with remainder is very difficult. In general, the β-base form must be reconstructed, then the operation performed and then the result factored again. The problem can be eased by extracting common factors from the arguments before the β-form is computed.

§3 Polynomial Operations

Now that we have looked at coefficient operations in the form of multiprecision integers we can proceed to consider polynomials over these coefficients. We note that other coefficient rings (e.g. the rationals) can be generated using the multiprecision integers. We will consider polynomials of the form:

$$f(x_1,\ldots,x_n) \in \mathcal{R}[x_1],\ldots,[x_n]$$

where $f(x_1,\ldots,x_n) = \sum_{j=1}^{m} a_i.x_i^{e_{i,j}}\ldots,x_n^{e_{n,j}}$

where the $\{e_{i,j}\}$ are the exponents of the indeterminants $\{x_i\}$. For the moment we will assume $e_{i,j} \in \underline{Z}^+$ (ie. are non-negative integers) but later we will weaken this requirement.

3.1) Background Considerations

We immediately consider multivariate polynomials because of the theorem from algebra;

Theorem 3.1.1: Polynomials over a ring from a ring.

This implies that if we have an algorithm for some ring operation on a k-variate polynomial then it can be generalised to a (k+1) variate polynomial by applying the algorithm recursively. Later we will have to specialize the rings somewhat to be able to perform other operations.

First we will look at polynomial forms.

3.1.1) Polynomial Forms.

The forms chosen for polynomials are cannonical so that a given polynomial has a unique representation. This means that operations on a polynomial produce a unique result giving a deterministic algorithm.

For the polynomial ring $\mathcal{R}[x_1] \ldots [x_n]$ the two most common cannonical forms are:

i) Recursive Canonical Form (RCF)

which exploits the isomorphism:

$$\mathcal{R}[x_1] \ldots [x_n] \simeq \mathcal{R}[x_1,\ldots,x_n][x_n]$$

The coefficients of the n-variate polynomials are polynomials in n-1 variables. This form is advantageous since many multivariate polynomial operations are recursive nature.

ii) Expanded Canonical Form (ECF)

exploits the isomorphism

$$\mathcal{R}[x_1] \ldots [x_n] \simeq \mathcal{R}[x_1,\ldots,x_n]$$

The numerical elements are coefficients of products of powers of variables. This form has the advantage that access to numerical coefficients is much easier than in RCF using the conventional structural representations.

The two forms are made canonical by imposing an ordering on the indeterminants and then ordering terms lexicographically.

For example:

The polynomial in RCF:

$$(3y^2 - (2z^3)y + 5z^2)x^2 + (4)x + ((-6z + 1)y^3 + 3y^2 + z^4 + 1)$$

is equivalent to the polynomial in ECF:

$$3x^2y^2 - 2x^2yz^3 + 5x^2z^2 + 4x - 6y^3z + y^3 + 3y^2 + z^4 + 1 .$$

Note that these are by no means the only canonical forms. Some other examples include:

iii) Consistant Degree Canonical Form (CDCF):

where the polynomial is considered as a sum of symmetric forms (ie. polynomials of the same total degree). Using our example in CDCF this would be:

$$(1) + (4x) + (3y^2) + (y^3) + (3x^2y^2 + 5x^2z^2 - 6y^3z + z^4) + (-2x^2yz^3)$$

or

iv) Factored Canonical Form:

If $\mathcal{R}$ is a unique factorisation domain then so is $\mathcal{R}[x]$ and so a polynomial can be uniquely factored into irreducible components. These components can be lexicographically ordered to give a canonical form. We should note that polynomial factoring is an expensive operation and so this form is similar to the factored form of integers in computational cost.

3.1.2) Models of Computation

To analyse algorithms for polynomial operations we assume that the polynomials conform to either of the following models:

i) The Dense Model:

This assumes that all coefficients of the polynomial are non zero. For a univariate polynomial of degree n:

$$a(x) = \sum_{j=0}^{n} a_j x^j$$

has n+1 non zero terms. For a v-variate polynomial:

$$a(\bar{x}) = \sum_{j_1=0}^{n_1} \cdots \sum_{j_v=0}^{n_v} a_{j_1 \cdots j_v} x_1^{j_1} \cdots x_v^{j_v}$$

has: $\prod_{i=1}^{v} (n_i+1) = (n+1)^v$ terms in it.

This model is the easiest one to analyse and so it is the usual one considered for the algorithms.

ii) The Sparse Model:

Assumes that most of the coefficients are non zero:

$$a(x) = \sum_{j=0}^{n} a_j x^{\alpha_j}$$

where $\alpha_{j+1} > \alpha_j$ for $n>j>0$.

This means that the polynomial is of degree $\alpha_n > n$, with n+1 terms.

In the multivariate case:

$$a(\bar{x}) = \sum_{j=0}^{m} a_j x_1^{\alpha_{j_1}} \ldots x^{\alpha_{j_v}}$$

the polynomial has m+1 terms. The sparse model generally further assumes that no cancellation or collection of common terms occurs during the operations on the polynomials. This means that the polynomial is very sparse i.e. a typical polynomial might be

$$a(x,y,z) = x + y + z \quad .$$

If the maximum exponent of the ith variable is e_i then we can uniquely encode the v-triple of exponents of a term in a sparse polynomial as:

$$\alpha_j = (\ldots((\alpha_{j_1} e_2 + \alpha_{j_2})e_3 + \ldots) e_v + \alpha_{j_v})$$

This is just Horner's rule applied to the exponents of a given term. This maps the sparse v-variate polynomial onto a sparse univariate polynomial. So without loss of generality we can consider sparse polynomials as univariate.

In fact these two models exhibit features of polynomials that are at either end of the spectrum of those seen in algebraic manipulation systems. Generally polynomials which are operated on are not completely dense however neither are they so sparse that no collection of terms occurs. So far a good intermediate model has not been developed.

Defn. 3.1.2: If $A(x_1,\ldots,x_r)$ is a polynomial in r variables we denote by $\partial_i(A)$ the degree of A in x_i and by $\partial(A)$ the degree vector $(\partial_1(A),\ldots,\partial_r(A))$. We shall often write deg(A) for $\partial_r(A)$ the degree of A in its main variable.

3.1.3) Polynomial Norms:

We can bound the size of coefficients produced in polynomial operations by use of polynomial norms on the coefficients.

Defn. 3.1.2: We can define the norms recursively:

a) For $A \in \underline{C}$, $|A|_1 = |A|_\infty = |A|$ (modulus)

b) For $A(x) \in \underline{C}[\bar{x}]$, $A(\bar{x}) = \sum_{i=0}^{m} A_i(\bar{x})\, x_n^i$

then $|A(x)|_1 = \sum_{i=0}^{m} |A_i|_1$

and $|A(x)|_\infty = \max_{0 \le i \le m} |A_i|_\infty$

So $|A(x)|_1$ is called the sum norm

and $|A(x)|_\infty$ " " " maximum norm

These norms have the usual properties of norms.

Lemma 3.1: These norms have the properties:

for $A,B \in \underline{C}[\bar{x}]$, $c \in \underline{C}$:

i) $|A| > 0$ unless $A=0$

ii) $|cA| = |c|\,|A|$

iii) $|A+B| \le |A| + |B|$.

Also we can show the following results.

Lemma 3.2: For $A,B \in \underline{C}[\bar{x}]$

$$|A \cdot B|_1 \le |A|_1 \cdot |B|_1$$

$$|A \cdot B|_\infty \le |A|_\infty \cdot |B|_1 \quad .$$

and

Lemma 3.3: For $A \in \underline{C}[\bar{x}]$. $A(\bar{x}) = \sum_{i=0}^{m.} A_i\, x_n^i$

$$|A| \ge |A_i| \quad .$$

and

Lemma 3.4: For $A \in \underline{C}[x]$, A univariate.

$$|A|_1 \geq (\sum_{i=0}^{m} A_i^2)^{1/2}$$

3.2) Representation of Polynomials

The data structure for a multivariate polynomial closely reflects the canonical form chosen for it. Most algebraic manipulation systems use only one canoncial form. The two most popular are:

a) Recursive Canonical Form:

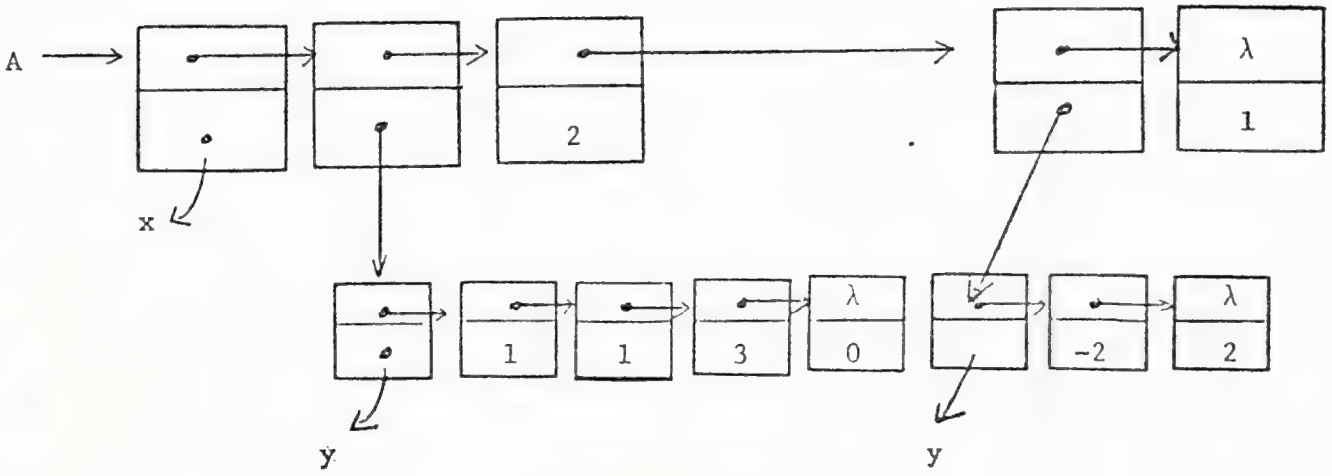
This is used by SAC-1, Reduce and MACSYMA. The form of a SAC-1 polynomial is a linked list of terms. Each term is represented by two list cells:

i) the first containing or pointing to the coefficient

ii) the second containing the exponent.

A list header cell contains a pointer to the runtime symbol table representation of the indeterminate.

e.g. the polynomial $A(x,y) = (y+3)x^2 + (-2y^2)x$ would be represented as:



In fact SAC-1 uses multi-precision integer coefficients and so the cells containing the integer coefficients (1, 3, -2) would in fact contain pointers to a linked list representing the coefficient. An alternative representation used by SAC-1 for polynomials with modular coefficients is to leave the exponents implicit and represent zero coefficients explicitly. Then the number of cells in the list is the degree +2 (including one list header). This leads to a representation close to that required by the dense model of computation but at some waste of storage. The other systems using RCF have very similar representations.

This representation makes storage allocation relatively easy since list cells can be allocated where needed and "garbage" collected using fairly standard algorithms. Using a system such as LISP which provides these mechanisms makes these operations even easier; which is one reason why MACSYMA and Reduce are based on LISP.

b) Expanded Canonical Form:

Altran is the most popular system which uses this representation, although as early as 1962 Williams described a system which used it. The polynomial A(x,y) in ECF would be:

$$A(x,y) = x^2 y + 3x^2 - 2xy^2$$

ECF then requires that each term be represented by the set of exponents of indeterminants in the polynomial. For reasons of storage economy it is usually chosen to pack them into one or more computer words. This means that the maximum value of the exponents must be known ahead of time.

Altran's data structure for a polynomial uses three arrays:

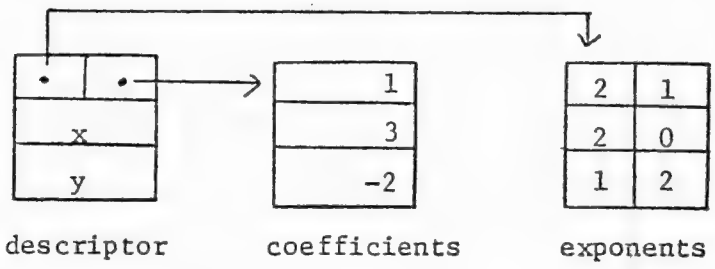
i) the descriptor block which contains information about the packing of the exponents and the external representation of the indeterminants

ii) a coefficient block which contains the numeric coefficients for each of the monomial terms in the polynomial.

iii) an exponent block which contains the exponents in packed form.

eg. for the polynomial $A(x,y) = x^2y + 3x^2 - 2xy^2$

the representation would be:



This structure provides a very compact representation which can be accessed very quickly since no chain of pointers need be followed. However the structure does give problems of storage allocation, especially when storage must be allocated for polynomials to be produced by some operations (eg. polynomial multiplication).

3.2) Polynomial Addition and Multiplication

We can now proceed to discuss algorithms for polynomial addition. For the sake of completeness we will write them in a quasi-algol notation.

3.2.1) The Primitive Operations

In our algorithms we will use a set of primitive operations. The exact form of these operations we will leave unspecified as it depends to a large extent what representation we have chosen for a polynomial. For example when we say list we may mean "linear list" (ie. array) or "linked list", this will depend on the structure.

The primitives are:

(a,b): concatenation, append list b to list a.

tail (a): yields the next element of the list a.

term (a): yields the term at the current head of list a.

coef (a): yields the coefficient of the term at the current head of the list a.

exp (a): gives the exponent of the term at the current head of list a.

N.B. This may be a single exponent or an exponent set, in which case addition of exponents would be pairwise.

new-term (c,e): yields a new polynomial term with coefficient c and exponent a.

3.2.2) Polynomial Addition:

Now the addition of two polynomials is just a simple merge of the elements, indexed by their exponents.

Algorithm 3.2.1: Add (a,b);

Input: the polynomials a,b .

Output: their sum c.

Steps:

1) Initialisation c:=0;

2) Iteration: while a≠0 and b≠0 do

```
if exp(a) > exp(b) then
begin c:=(c,term(a)); a:=tail(a); end
else if exp(b) > exp(a) then
begin c:=(c,term(b)); b:=tail(b); end

else begin
     t:= add(coef(a), coef(b));
     if t≠0 then c:=(c,new-term(t,exp(a)));
     a:=tail(a); b:=tail(b);
     end;
```

3) Completion:

```
if a≠0 then c:=(c,a)
else if b≠0 then c:=(c,b);
return (c);
```

□

Add may call itself recursively if the polynomials are in RCF.

We see that if there are n terms in a and m terms in b then the algorithm may perform ~ (m+n) operations to form the sum.

More specifically

Lemma 3.2.1: If $a_0 = |a|_\infty$, $b_0 = |b|_\infty$

and $m_i = \partial_i(a)$, $n_i = \partial_i(b)$.

then

$$T_{Add} \leq (L(a_0) + L(b_0)) \prod_{i=1}^{r} (m_i + n_i + 1) .$$

We can apply a result of Karp's about general merging to our situation:

Theorem 3.2.2: There exist polynomials a,b such that at least m+n comparisons must be made to find their sum.

As a corollary we note that the number of terms in the sum is given by:

Corollary 3.2.3: If $m \geq n$ then there are at most m+n, at least m-n and at least max (m,n) (if no cancellation occurs) terms in the sum.

3.2.3) Polynomial Multiplication

We can use the addition algorithm to specify a polynomial long multiplication algorithm.

Algorithm 3.2.2: Mult (A,B);

Input: the polynomials A,B;

Output: their product C;

Steps:

1) Initialisation C:=0;

2) Iteration: while a≠0 do begin

B1 := B; C1 := 0;

3) Partial while B1 ≠ 0 do begin

including a test t:= mult(coef(A), coef(B1);

for divisors of zero: if t≠0 then C1:=(C1, new-term (t, exp(A) + exp(B1)))

B1:=tail (B1);

end;

C := Add(C,C1); A := tail(A);

end;

4) Completion: return (C); □

Note that mult may call itself recursively if the polynomials are in RCF.

Analysis:

In the dense univariate case if deg(A) = n and deg(B) = m then this algorithm uses ≤ mn arithmetic operations. In particular we have

Theorem 3.2.4: If $m_i = \partial_i(A)$ and $n_i = \partial_i(B)$ and $a_0 = |A|_\infty$, $b_1 = |B|_1$. then

$$T_{Mult} \leq L(a_0)\ L(b_1) \prod_{i=1}^{r} (m_i+1)(n_i+1) .$$

For the sparse model we have:

Lemma 3.2.5: The classical multiplication algorithm can be cubic in the number terms

Proof: If A has n terms and b has m terms $n \leq m$ then the final result can have mn terms.

At the k-th iteration the partial product can have km terms and so the addition of partial product and new sum uses (km+m) comparisons. Thus the total for the algorithm is:

$$T_{s\ mult} \leq \sum_{k=1}^{n} (k+1)m \sim n^2 m$$

$$\sim n^3 \quad \text{if } n=m \qquad \square$$

What is slowing us down is the addition steps. Notice that the problem is the sequencing of the steps. We could perform a sparse polynomial multiplication in T ≤ nm log n steps using the following algorithm.

1) Compute all nm cross product terms.

2) Sort them by their exponents in T ≤ nm log n steps

This gives us a $T \leq n^2$ log n algorithm. We can derive algorithms with such time bounds in other ways.

3.2.4) Sparse Polynomial Multiplication

The general method for multiplying sparse polynomials is to be careful how we do the additions. One method we can use:

I) Divide and Rule Method:

We can multiply:

$$\left(\sum_{j=1}^{n} a_j x^{\alpha_j}\right) \cdot \left(\sum_{j=1}^{m} b_j x^{\beta_j}\right) = \left(\sum c_j x^{\lambda_j}\right)$$

as the form:

$$\left(\sum_{j=1}^{n/2} a_j x^{\alpha_j}\right) \cdot \left(\sum_{j=1}^{m} b_j x^{\beta_j}\right) + \left(\sum_{j=n/2+1} a_j x^{\alpha_j}\right) \cdot \left(\sum_{j=1}^{m} b_j x^{\beta_j}\right)$$

(ie. divide the multiplication into 2 equal parts, recursively). Then we have:

Theorem 3.2.6: The divide and rule method works in

$$T_{DR}(mn,)S \leq \quad mn \log n \text{ steps.}$$

Proof. From the form of the method we can derive the recurrence relation:

$$T_{DRS}(m,n) = 2\, T_{DRS}(m,n/2) + T_A\left(\frac{mn}{2}, \frac{mn}{2}\right)$$

$$= 2\, T_{DRS}(m,n/2) + mn$$

Expanding the recurrence relation we get:

$$T_{DRS}(m,n) = \sum_{i=1}^{\log n} mn + 2^{\log n}\, T(m,1)$$

$$\leq mn \log n \quad .$$

In the dense model we would not get mn/2 terms from each sub-product but m+n/2 terms. Hence its computing time is:

$$T_{DRD}(m,n) = 2\, T_{DRD}(m,n/2) + 2(m+n/2)$$

$$= \sum_{i=1}^{\log n} (2m + n) + 2^{\log n}\, T_{DRD}(m,1)$$

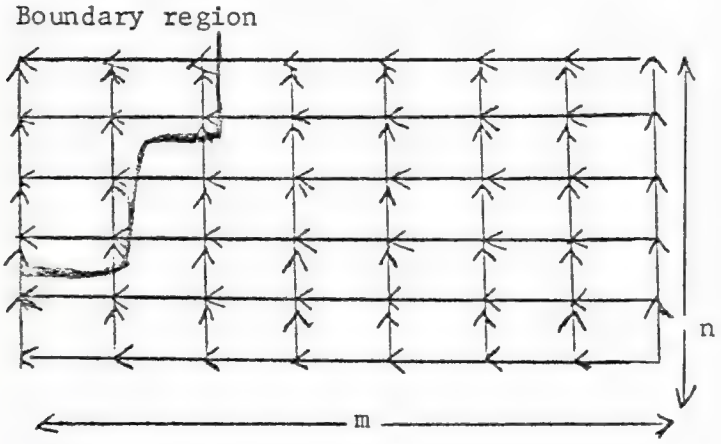
$$\leq mn \qquad \square$$

In other words the divide and rule method would run in the same time as the classical algorithm for the dense case. This is a property which might not be enjoyed by the sorting method.

It might appear that the recursive form of the divide and rule method might be difficult to implement when a list representation for polynomials is used. However the algorithm can be converted to a iterative form by storing partial products on a stack and merging them together in such a fashion that sums of equal length are added together.

II) Johnson's Algorithm:

We can be clever about the order in which we generate the terms in the product. Consider the cross-sum table of exponents of two polynomials A(x) and B(x). This table will be ordered along each row and each column since the terms in both polynomials are ordered by exponent.



cross sum

table of exponents

If we can generate the terms of the product in order then at the kth step all the terms in a region of the table from the top left hand end will be in the product and the next element to be added to the product will be chosen from those terms on the boundary of the region.

There can be at most n such terms on the boundary. The trick is in knowing which one to choose next. Johnson suggests storing the boundary elements in a heap so that they will be ordered on entering them and the largest can be quickly retrieved. In fact there are a number of equivalent such structures (eg. AVL trees or 2-3 trees). We will think of the structure as a priority queue which can be manipulated by the primitives:

queue (a,b): element b is inserted into the queue ordered with respect to a. a is the exponent of the new term and b is a 3-tuple <i,j,c> where i and j are pointers to the terms in the two

polynomials which produced the new term and c is the coefficient of the new term. Duplicate entries are discarded.

top (a,b): extracts the largest element of the queue with respect to the a's.

We assume that we can test if the queue is empty to terminate the algorithm. Also we introduce additional primitive operations to manipulate the last (lowest exponent) term in the product. These are necessary since we generate the terms in the product by decreasing exponent. To catch zero coefficients we need:

last_coef (A): which accesses the last coefficient of a polynomial A.

and

drop_last (A): which drops the last term of polynomial A.

in a list structured these might be implemented by keeping track of the term before the last one.

Algorithm 3.2.3: Mult (U,V);

Input: the polynomials U,V;

Output: their product C;

Steps:

```
1) Initialisation:    C:=0; t=mult(coef(U), coef(V)); λ1:=0;
                      queue (exp(U) + exp(V), <U,V,t>);
2) Iteration:         while queue not empty begin
                      top (<u,v,t>);
                      if λ1≠λ then C:=(C, new-term (t,λ))
                      else begin
                         t:=Add (t, last_coef (C));
                         if t≠0 then last_coef (C):=t
                         else begin drop-last (C); λ:=0 end
                              end;
                      λ1:=λ;
```

3) Fill queue with elements on the boundary:

```
if U ≠ 0 then begin
U:=tail (U); t:=mult (coef(U), coef (V));
if t≠0 then queue (exp(U) + exp(V), <U,V,t>);
end;
if V≠0 then begin
V:=tail (V); t:=mult (coef (U), coef (V));
if t≠0 then queue (exp(U) + exp(V), <U,V,t>);
   end;
end;
```

4) Completion:

```
return (C)    ;
```

□

As an example of the algorithm we consider the multiplication of $u = x^1 - 2x^3 + 3x$ by $V = 4x^6 + 2x^2 + 1$.

We use the numerical index of the term as a pointer to it ie. <1,1,4> indicates the first term in each polynomial.

Iterations	Queue contents
0) C = 0, t = 4, λ1 = 0	|13, <1,1,4>|
1) λ = 13, C = $4x^{13}$, λ1 = 13	|9, <2,1,-8> 9, <1,2,2>|
2) λ=9, t=-8, C=$4x^{13} - 8x^9$	|9, <1,2,2> 7, <3,1,12> 5, <2,2,-4>|
3) λ=9, t=2, C=$4x^{13} - 6x^9$	|7, <1,3,1> 7, <3,1,12> 5, <2,2,-4>|

4) $\lambda=7,\ t=1,\ C=4x^{13} - 6x^{9} + x^{7}$

7, <3,1,12>
5, <2,2,-4>
3, <2,3,-2>

5) $\lambda=7,\ t=12,\ C=4x^{13} - 6x^{9} + 13x^{7}$

5, <2,2,-4>
3, <3,2,6>
3, <2,3,-2>

6) $\lambda=5,\ t=-4,\ C=4x^{13} - 6x^{9} + 13x^{7} - 4x^{5}$

3, <3,2,6>
3, <2,3,-2>

7) $\lambda=3,\ t=6,$
$C=4x^{13} - 6x^{9} + 13x^{7} - 4x^{5} + 6x^{3}$

3, <2,3,-2>
1, <3,3,3>

8) $\lambda=3,\ t=-2$
$C=4x^{13} - 6x^{9} + 13x^{7} - 4x^{5} + 4x^{3}$

1, <3.3,3>

9) $\lambda=1,\ t=3,$
$C=4x^{13} - 6x^{9} + 13x^{7} - 4x^{5} + 4x^{3}$

empty

Theorem 3.2.7: The queued multiplication algorithm works in

$$T_{PMS} \leq nm \log n$$

for polynomials of m and n terms, $m \geq n$.

Proof: At most $\leq \log n$ operations are needed to insert elements into a priority queue (or a heap or AVL tree etc.). There may be mn terms in the product. This gives mn log n operations.

Note that this algorithm can also work well with dense polynomials. However it is necessary to alter the queue so that it maintains a list of terms with equal exponents as a queue member. In the dense case because of the order in which terms are processed there will be at most 2 members of the queue. This gives a time of $T_{QMS} \leq mn$.

□

It is an open question if mn log n operations are necessary for sparse polynomial multiplication. There are rumors that it can be done in mn steps but I know of no algorithm which works in that time.

Finally we note that the algorithms given here for polynomial ring operations have assumed that the exponents e_i of the variables are non-negative integers. However we have only used the fact that exponents are linearly ordered ie. for three exponents e_i, e_j, e_k the following conditions hold:

i) One and only one of the relations $e_i < e_j$, $e_i = e_j$, $e_i > e_j$ is true.

ii) If $e_i < e_j$ and $e_j < e_n$ then $e_i < e_k$.

This means that these algorithms will work equally well for exponents drawn from any linearly ordered set. In particular we could allow exponents drawn from:

i) Z : the set of all integers

ii) Q : the set of rational numbers

or even

iii) R[x]: the set of polynomials where the coefficient ring is linearly ordered. Such polynomials can be lexicographically ordered with respect to their degrees and coefficients.

Such domains are outside the usual definitions of polynomials but they are still valid rings for which the algorithms work.

Some Specific GCD Algorithms

We can best illustrate the effects of specific algorithms by way of an example. Suppose we wished to compute the GCD of, A, B $\in \mathbb{Z}[x]$.

$A(x) = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5$

$B(x) = 3x^6 + 5x^4 - 4x^2 - 9x + 21$.

1) Euclid's Algorithm:

We might work in $\mathbb{Q}[x]$ in order to get an EUD and apply Euclid's Algorithm. This gives us the following Polynomial Remainder Sequence (PRS). (with powers of x removed).

1, 0, 1, 0, -3, -3, 8, 2, -5

3, 0, 5, 0, -4, -9, 21

-5/9, 0, 1/9, 0, -1/3

$-\frac{117}{25}$, -9, $\frac{411}{25}$

$\frac{233150}{6591}$, $-\frac{102500}{2197}$

$\frac{1288744821}{543589225}$

So the A and B are relatively prime.

Note the growth in the size of the coefficients. If we so wished we could apply the same algorithm to multivariate calculations in $\mathbb{Z}[x_1,\ldots,x_n]$ by working in $\mathbb{Z}(x_1,\ldots,x_{n-1})[x_n]$. This means that we would have to apply the GCD algorithm recursively to perform the arithmetic in $\mathbb{Z}(x_1,\ldots,x_{n-1})$. It also means that the coefficients become rational functions and their growth produces serious problems.

It can be improved somewhat by making each element of the sequence monic.

i.e.

1, 0, 1, 0, -3, -3, 8, 2, -5

1, 0, 5/3, 0, -4/3, -3, 7

1, 0, -1/5, 0, 3/5

1, $\frac{25}{13}$, $-\frac{49}{13}$

1, $-\frac{6150}{4663}$

1

But we are still left with having to do GCD's to perform the coefficient operations.

II Euclidean PRS Algorithm:

To avoid doing coefficient GCD's we remember the pseudo-diversion construction which enables us to do all arithmetic over the original domain $\mathbb{Z}$ instead of working in the quotient field $\mathbb{Q}$. This gives us the PRS:

1, 0, 1, 0, -3, -3, 8, 2, -5

3, 0, 5, 0, -4, -9, 21

-15, 0, 3, 0, -9

15795, 30375, -59535

1254542875143750, - 1654608338437500

1259333879550074310093114199218750 0

However the coefficients grow even more drastically than before. In fact this has been described as the WW (world's worst) GCD algorithm. The only comparably bad method would be to work over the quotient field but not apply GCD's recursively to the coefficients. This coefficient growth is one of the rocks on which early algebraic manipulation systems floundered. In fact we have:

Lemma: The coefficients in an Euclidean PRS can grow exponentially

in n the degree of the input polynomials.

Proof: Let

$$|A|_\infty , |B|_\infty \leq N$$

Also in the PRS let the degree of successive elements change only by 1.

i.e. $\deg(a_i) = \deg(a_{i+1}) + 1$

Then applying psuedo division at the k th iteration, we will have coefficients in a_k of size N^{ℓ_k}. To get the next remainder we must multiply a_{k+1} through by an element of size $(N^{\ell_k})^2$. Thus the coefficients in the remainder can be of size $N^{\ell_{k+2}}$ where

$$\ell_{k+2} = 2\ell_k + \ell_{k+1}$$

Solving the recurrence $\ell_n \simeq (1 + \sqrt{2})^n/2$

so the coefficent of the nth remainder may be of size

$N^{\frac{(1+\sqrt{2})^n}{2}}$. □

3.4 Some Notation

We will introduce the following notation for the euclidean algorithm and its variants. Given $F_0(x)$ and $F_1(x)$ we are computing the PRS $\{F_0,F_1,\ldots,F_k\}$ in such a manner that F_{i+1} is similar to the pseudo-remainder of F_{i-1} (pseudo-) divided by F_i. ie.

$$\alpha_i F_{i-1} = Q_i F_i + \beta_i F_{i+1}$$

where α_i and β_i are chosen to from the ID of the coefficients. $F_k(x)$ is similar to the GCD of F_0 and F_1.

Also we will have: $L(F_i) = f_i$ and $\delta_x(F_i) = n_i$.

Therefore by the pseudo-division construction:

$$L_i = f_i^{\,n_{i-1}-n_i+1} \quad .$$

The differences in the following algorithms rest on the choice and computation of β_i so as to minimise the growth of coefficients at each step.

III. Primitive PRS Algorithm.

As alternative is to compute the primitive part of each remainder as it is produced. For our example this gives:

1, 0, 1, 0, -3, -3, 8, 2, -5

3, 0, 3, 0, -4, -9, 21

5, 0, -1, 0, 3

13, 25, -49

4663, -6150

1

Using our notation this means:

$$\beta_i = \text{cont}\,(F_{i+1}) \quad .$$

This gives us an algorithm with minimal coefficient growth and illustrates the

fact that the coefficient growth is produced by large contents generated by the pseudo-division. However we must again apply the GCD recursively to compute the contents, an expensive operation.

III. The Reduced PRS Algorithm.

It can be shown (Brown & Traub) using the theory of subresultants that a valid choice of β_i is:

$$\beta_3 = 1, \qquad \beta_i = \alpha_{i-1}, \qquad 4 \le i \le k.$$

This absorbs some of the growth of the coefficients. In our example:

1, 0, 1, 0, -3, -3, 8, 2, -5

3, 0, 5, 0, -4, -9, 21

-15, 0, 3, 0, -9

585, 1125, -2205

-18885150, 24907500

527933700

It can be shown that if the PRS is normal ie. $n_{i-1} - n_i = 1$, $1 \le i \le k$ and so $n_i = n_0 - i$ then the coefficient of similarity is 1. This means we can bound the coefficients of the i-th element of the PRS.

IV. The Subresultant PRS:

The Reduced PRS algorithm works reasonably well if the PRS is normal however if it is abnormal then the coefficients can start growing exponentially again. To handle this situation Collin's developed the Subresultant PRS algorithm. The algorithm divides out the relavent subresultants from the coefficients as necessary. It's effectiveness can be shown using the resultant constructions.

The coefficients chosen are:

$$\beta_3 = (-1)^{\delta_i+1} \quad , \quad \beta_i = -f_{i-2}\psi_i^{\delta_{i-2}} \qquad 4 \le i \le k$$

where $\delta_i = n_i - n_{i-1}$ and

$$\psi_3 = -1, \quad \psi_i = (-f_{i-2})^{\delta_{i-3}} \quad \psi_{i-1}^{1-\delta_{i-3}}$$

in our example this gives.

1, 0, 1, 0, -3, -3, 8, 2, -5

3, 0, 5, 0, -4, -9, 21

15, 0, -3, 0, 9

65, 125, -245

9326, -12300

260708

which differs from the reduced PRS only at the last step.

V. The Improved PRS:

Further improvements can be made to this touch of dividing out redundant coefficients factors. Sometimes it is possible to choose a τ_i which can be divided from the F_i. Although there is no known algorithm for finding such a τ_i other than computing the content a simple strategy is:

$$\text{choose} \quad \tau_i \begin{cases} f_{i-1} & \text{when } f_{i-1} \mid F_i \\ = 1 & \text{otherwise.} \end{cases}$$

This makes our example.

1, 0, 1, 0, -3, -3, 8, 2, -5

3, 0, 5, 0, -4, -9, 21

5, 0, -1, 0, 3

13, 25, -49

9326, -12300

260708

which gives further improvement in reducing the coefficient growth.